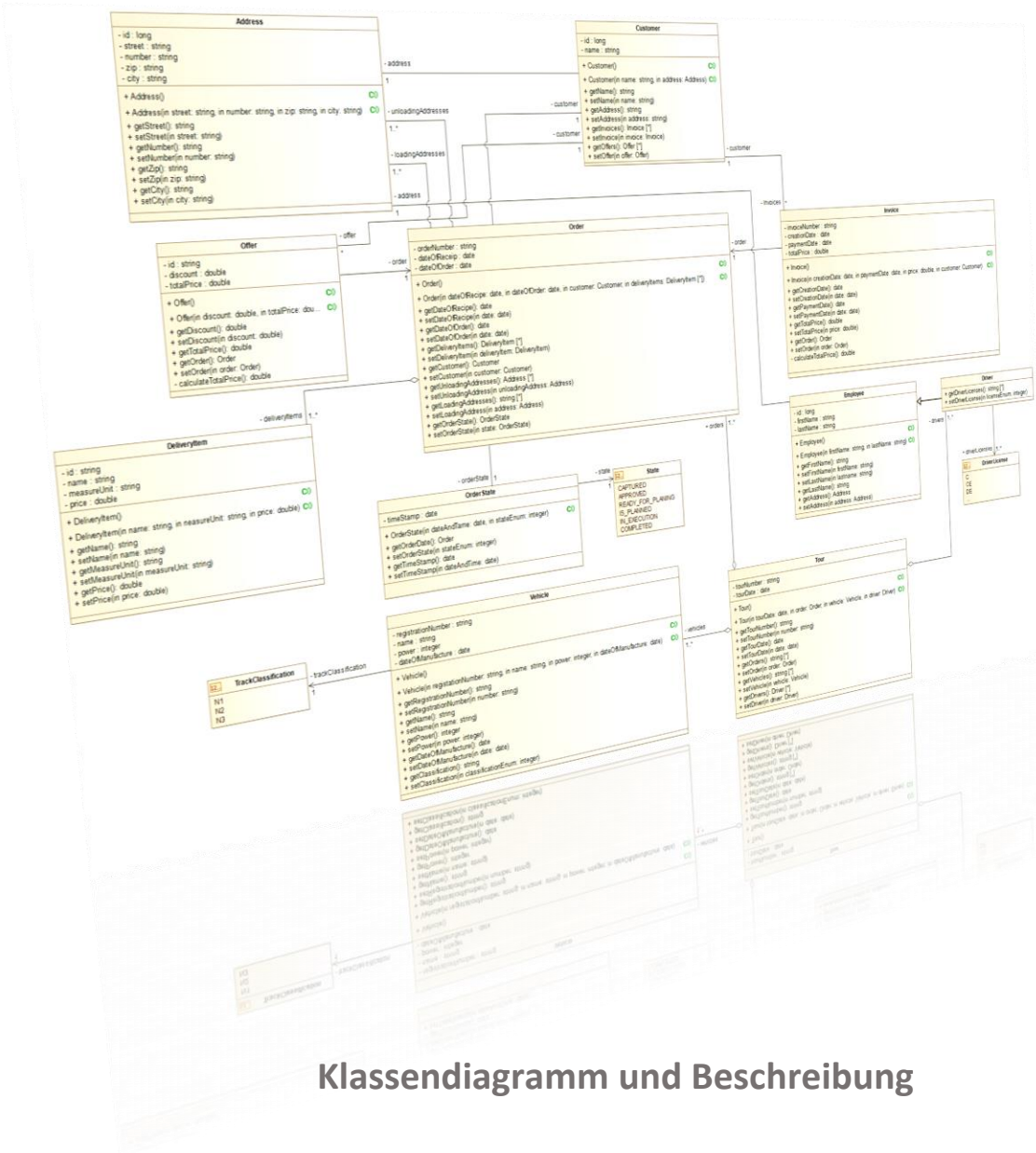
# Ausarbeitung zum Thema
# „Software für ein Speditionsunternehmen“




## Klassendiagramm und Beschreibung

# Inhalt

# 1 Einführung.

Die von Use-Case-Diagramm und Use-Case-Beschreibungen abgeleitete Klassendiagramm unterteilt sich in Modelio – Projekt in drei Packages. Es handelt sich um die Packages Models, Repositories und Services. „Models“ beinhaltet alle verwendete Entities, sowie zeigt deren Zusammenhang.
Beim „Services“ und „Repositories“ handelt es sich um die Kontrollklassen wobei „Services“ für den späteren Einsatz bei z.B. GUI oder „Controllers“, je nach potenziellen Entwurfsmuster, vorgesehen ist. „Repositories“ stellt eine Datenquellenanbindung für die „Services“ dar.

# 2 Package „Models“.

## 2.1 Entities.

Ausgehend aus Use – Case – Diagramm und dazugehörigen Beschreibungen für die Anwendungsfälle, von dem ersten Teil der Portfolio Aufgaben, wurde als erstes überlegt welche Objektmodele als Entities dargestellt werden müssen und in welcher Verbindung sie zueinanderstehen sollen. Für den Entwurf der Modelle wurde es nach Ablauf der Anwendungsfälle, allen Aktoren des Auftragsdispositionssystems gegangen. Zusammengefasst, nach der Annahme eines Angebots bekommt der Verkäufer ein Auftrag von Stammkunden oder neuen Kunden. Im Falle, wenn es sich um Neukunden handelt, muss ein Kunde im System eingetragen werden. Selbstverständlich besitzt der Kunde auch ein Standort (für die Vereinfachung wurde entschieden, dass ein Kunde nur eine Kontaktadresse hat und unter einer Adresse nur ein Kunde besiedelt ist) also muss auch eine Adresse vorhanden sein. Zum Schluss gibt der Verkäufer, den Auftrag frei. Daraus ergibt sich, dass die Use Cases des Verkäufers, folgende Entity-Klassen benötigt werden:

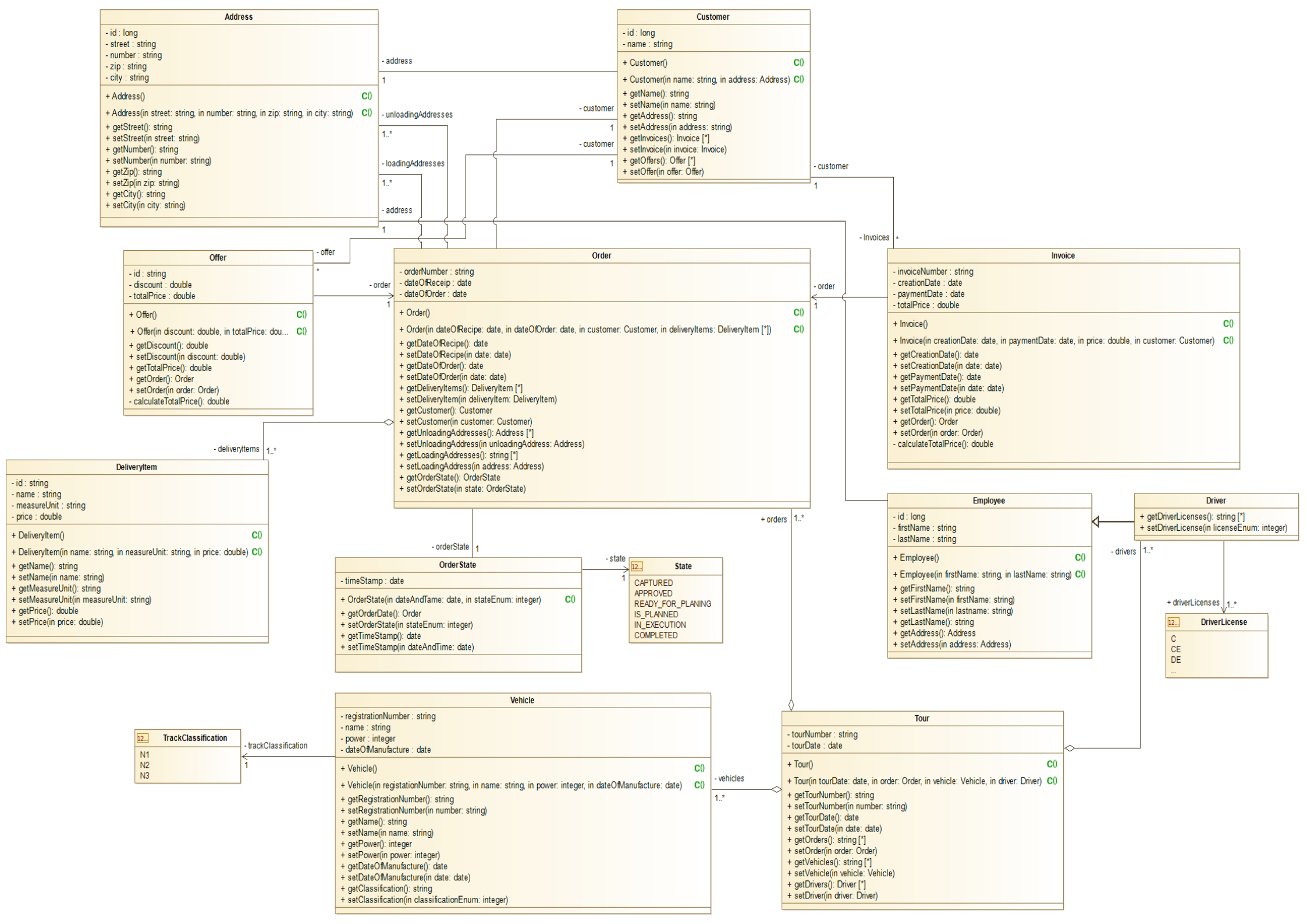
- Angebot (Offer)
- Auftrag (Order)
- Kunde (Customer)
- Adresse (Address)

Ein Disponent soll den freigegebenen Auftrag einplanen, dafür muss er das Vorhandensein von passendem Fahrzeug/e und Fahrern prüfen. Für den Fall, wenn eigene Ressourcen fehlen, wird eine Anfrage an Subunternehmen gesendet. Antwortet der Subunternehmer auf Anfrage negativ, wird der Auftrag zurückgewiesen. Für die Anwendungsfälle der Disponenten entstehen dann Entities:

- Fahrzeug (Vehicle)
- Fahrer (Driver)
- Subunternehmen (Partner)*

Ist eine Überprüfung erfolgreich und ein Auftrag ist eingeplant, muss ein LKW-Fahrer seine Tour durchführen, dafür muss er auch passende Fahrerlaubnis für die bestimmte Fahrzeugklasse besitzen. Die Kandidaten Fahrerlaubnis (License) und Fahrzeugklassen (TrackClassification) werden nicht wie die Entities verwendet, sondern als Enumerators für die Entities „Driver“ und „Vehicle“ verwendet. Nach dem ein LKW-Fahrer sein Auftrag erfolgreich durchgeführt hat kann das Back Office eine Rechnung für den Kunden erstellen und somit eine Auftragsabwicklung abschließen. Dabei ist zu beachten, dass der Auftragsstatus sich ausgehend von Bearbeitungsphase ändert und deswegen soll auch als ein Enumerator für die Modellklasse „OrderState“ verwendet werden. Eine Rechnung soll als Entity „Invoice“ erfasst werden.

*Das Subunternehmen (Partner) wird in Klassendiagramm nicht verwendet, da auch in Use - Case - Teil auf Anwendungsfälle für Subunternehmen verzichtet wurde. Jedoch bei Bedarf, aufgrund des Klassendiagram Entwurfes soll es auch kein Problem darstellen Subunternehmen für
Entity- und Kontrollklassen einzusetzen.

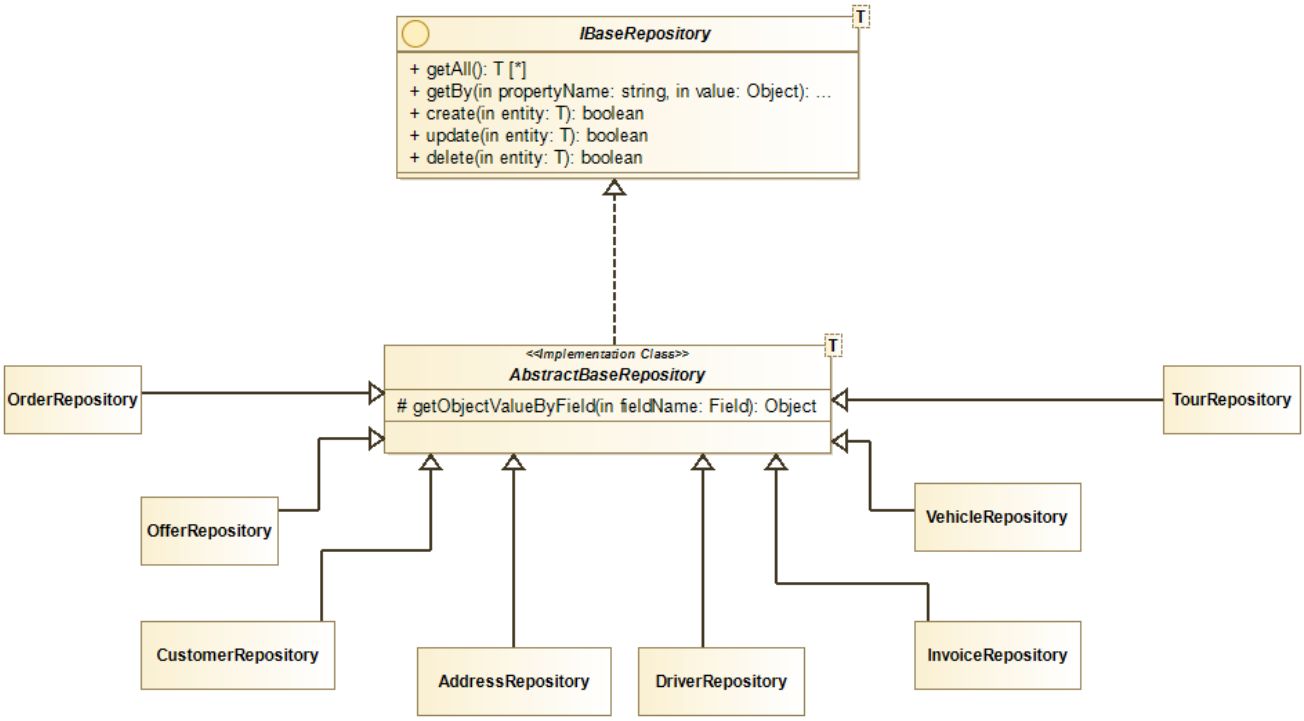

Address
- id : long
- street : string
- number : string
- zip : string
- city : string
+ Address()
+ Address(in street: string, in number: string, in zip: string, in city: string)
+ getStreet(): string
+ setStreet(in street: string)
+ getNumber(): string
+ setNumber(in number: string)
+ getZip(): string
+ setZip(in zip: string)
+ getCity(): string
+ setCity(in city: string)
Customer
- id : long
- name : string
+ Customer()
+ Customer(in name: string, in address: Address)
+ getName(): string
+ setName(in name: string)
+ getAddress(): string
+ setAddress(in address: string)
+ getInvoices(): Invoice [*]
+ setInvoice(in invoice: Invoice)
+ getOffers(): Offer [*]
+ setOffer(in offer: Offer)
- address
1
- unloadingAddresses
1..*
- loadingAddresses
1..*
- address
1
- customer
1
- customer
1
- customer
1
- Invoices
*
Offer
- id : string
- discount : double
- totalPrice : double
+ Offer()
+ Offer(in discount: double, in totalPrice: dou...
+ getDiscount(): double
+ setDiscount(in discount: double)
+ getTotalPrice(): double
+ getOrder(): Order
+ setOrder(in order: Order)
- calculateTotalPrice(): double
- offer
*
- order
1
Order
- orderNumber : string
- dateOfReceip : date
- dateOfOrder : date
+ Order()
+ Order(in dateOfRecipe: date, in dateOfOrder: date, in customer: Customer, in deliveryItems: DeliveryItem [*])
+ getDateOfRecipe(): date
+ setDateOfRecipe(in date: date)
+ getDateOfOrder(): date
+ setDateOfOrder(in date: date)
+ getDeliveryItems(): DeliveryItem [*]
+ setDeliveryItem(in deliveryItem: DeliveryItem)
+ getCustomer(): Customer
+ setCustomer(in customer: Customer)
+ getUnloadingAddresses(): Address [*]
+ setUnloadingAddress(in unloadingAddress: Address)
+ getLoadingAddresses(): string [*]
+ setLoadingAddress(in address: Address)
+ getOrderState(): OrderState
+ setOrderState(in state: OrderState)
- order
1
Invoice
- invoiceNumber : string
- creationDate : date
- paymentDate : date
- totalPrice : double
+ Invoice()
+ Invoice(in creationDate: date, in paymentDate: date, in price: double, in customer: Customer)
+ getCreationDate(): date
+ setCreationDate(in date: date)
+ getPaymentDate(): date
+ setPaymentDate(in date: date)
+ getTotalPrice(): double
+ setTotalPrice(in price: double)
+ getOrder(): Order
+ setOrder(in order: Order)
- calculateTotalPrice(): double
- deliveryItems
1..*
DeliveryItem
- id : string
- name : string
- measureUnit : string
- price : double
+ DeliveryItem()
+ DeliveryItem(in name: string, in neasureUnit: string, in price: double)
+ getName(): string
+ setName(in name: string)
+ getMeasureUnit(): string
+ setMeasureUnit(in measureUnit: string)
+ getPrice(): double
+ setPrice(in price: double)
- orderState
1
OrderState
- timeStamp : date
+ OrderState(in dateAndTame: date, in stateEnum: integer)
+ getOrderDate(): Order
+ setOrderState(in stateEnum: integer)
+ getTimeStamp(): date
+ setTimeStamp(in dateAndTime: date)
- state
1
State
CAPTURED
APPROVED
READY_FOR_PLANING
IS_PLANNED
IN_EXECUTION
COMPLETED
+ orders
1..*
Employee
- id : long
- firstName : string
- lastName : string
+ Employee()
+ Employee(in firstName: string, in lastName: string)
+ getFirstName(): string
+ setFirstName(in firstName: string)
+ setLastName(in lastname: string)
+ getLastName(): string
+ getAddress(): Address
+ setAddress(in address: Address)
Driver
+ getDriverLicenses(): string [*]
+ setDriverLicense(in licenseEnum: integer)
- drivers
1..*
+ driverLicenses
1..*
DriverLicense
C
CE
DE
...
TrackClassification
N1
N2
N3
- trackClassification
1
Vehicle
- registrationNumber : string
- name : string
- power : integer
- dateOfManufacture : date
+ Vehicle()
+ Vehicle(in registationNumber: string, in name: string, in power: integer, in dateOfManufacture: date)
+ getRegistrationNumber(): string
+ setRegistrationNumber(in number: string)
+ getName(): string
+ setName(in name: string)
+ getPower(): integer
+ setPower(in power: integer)
+ getDateOfManufacture(): date
+ setDateOfManufacture(in date: date)
+ getClassification(): string
+ setClassification(in classificationEnum: integer)
- vehicles
1..*
Tour
- tourNumber : string
- tourDate : date
+ Tour()
+ Tour(in tourDate: date, in order: Order, in vehicle: Vehicle, in driver: Driver)
+ getTourNumber(): string
+ setTourNumber(in number: string)
+ getTourDate(): date
+ setTourDate(in date: date)
+ getOrders(): string [*]
+ setOrder(in order: Order)
+ getVehicles(): string [*]
+ setVehicle(in vehicle: Vehicle)
+ getDrivers(): Driver [*]
+ setDriver(in driver: Driver)

### 2.2 Beziehungen und Anwendung der Analysemustern.

Ein Angebot und eine Rechnung, können nur für einen Kunden erstellt werden, gleichzeitig kann ein Kunde mehrere Angebote und/oder Rechnungen haben. Dasselbe gilt auch für die Beziehung Kunde Auftrag, ein Auftrag wird von einem Kunden erstellt und bezieht sich zu diesem Kunden, gleichzeitig kann ein Kunde ein Unternehmen mit mehreren Lieferungen beauftragen. Für die Beziehungen Customer-Invoice, Customer-Offer, Customer-Order entsteht dann eine 1:N Beziehung

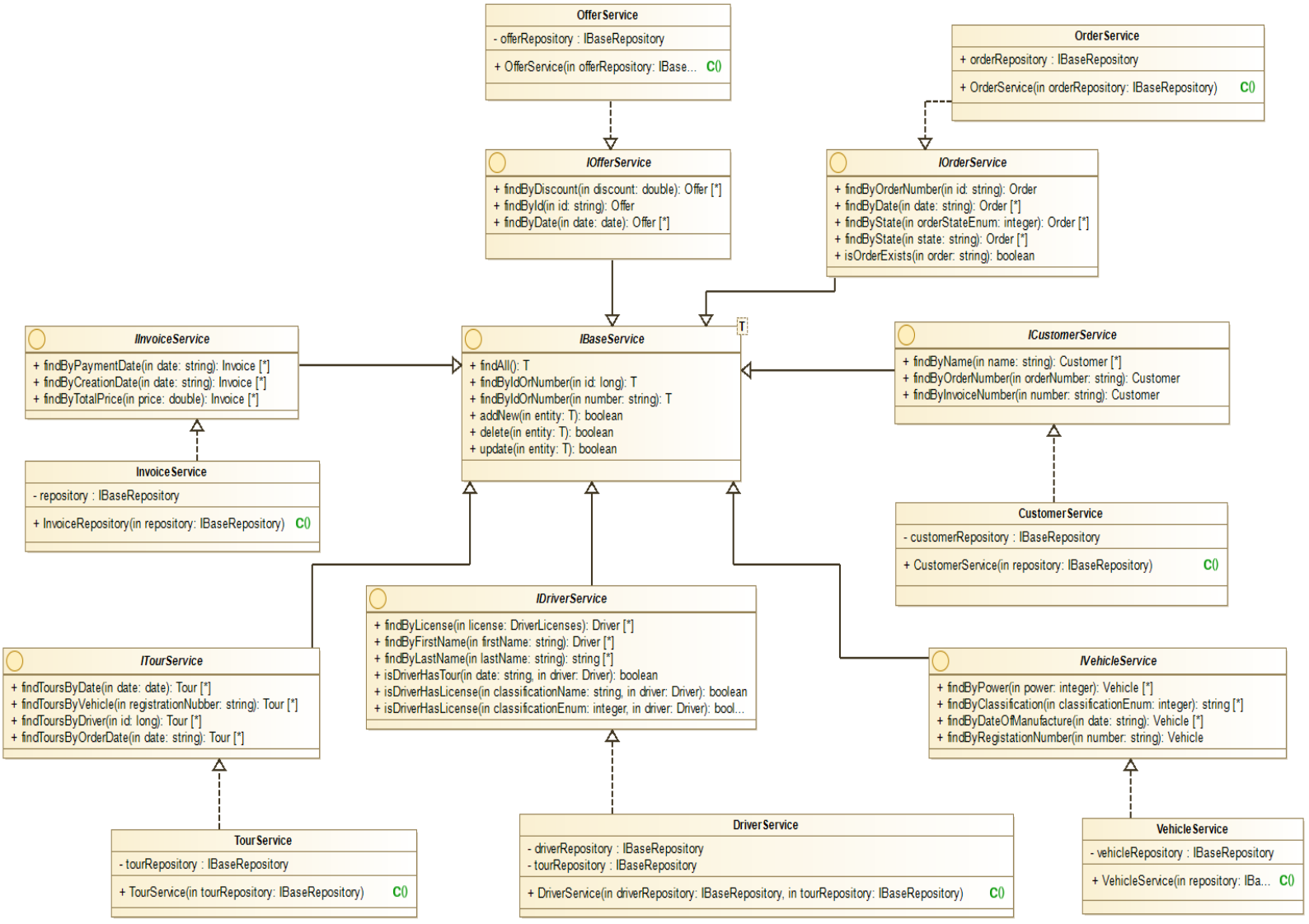
Die Beziehung Order – OrderState geht aus der Analysemuster „History“ aus. Diese Beziehung stellt ein Auftragsstatus eines Auftrages zu bestimmten Zeitpunkt dar und somit besteht eine Möglichkeit Auftragsstatus zu bestimmten Zeitpunkt abzulesen.
Ein Auftrag kann mehrere Be- und Entladeorte haben und es wird angenommen, dass diese Be- und Entladeorte nur zu einem Auftrag sich beziehen können. Dadurch entsteht eine 1:N Beziehung zwischen „Order“ und „Address“ Entities.
Für die Beziehungen Customer-Address und Employee-Address wurde eine 1:1 Beziehung angewandt. Für den zu bearbeitenden Fall wird auf Möglichkeit, dass ein Unternehmen z.B. mehrere Niederlassungen oder ein Mitarbeiter Zweitwohnung besitzt, verzichtet (obwohl es sinnvoll wäre).
Ein Analysemuster „Baugruppe“ wurde für das Zusammensetzen einer Tour angewandt. Eine Tour setzt sich zusammen aus Fahrzeug/e (Vehicles), zu ausführendem Auftrag/Aufträge (Orders) und einen Fahrer/n (Drivers), die diese Tour ausführen. Gleichzeitig können eins oder mehrere Fahrer, Fahrzeuge, Aufträge an einer Tour beteiligt werden. Dadurch kommt man zu einer 1:N Beziehung für Tour-Driver, Tour-Order, Tour-Vehicle und gleichzeitig zu einem Analysemuster „Liste“, für jede einzelne Beziehung von Tour.

## 3 Package „Repositories“



Bei dem Package „Repositories“ handelt es sich um eine schematische Realisierung des Repository Patterns für den späteren Einsatz der Repositorien bei dem Package „Services“. Um die Repositorien für die davor erstellte Modellen zu realisieren, ist ein Import des Packages „Models“ notwendig. Da die Repositorien grundsätzlich über die Grunddatenoperationen (Lesen, Schreiben, Aktualisieren, Löschen) verfügen sollen, ist es sinnvoll als Erstes ein generisches Interface zu entwerfen und dieses mit einer abstrakten Klasse zu implementieren. Die Methode „findBy“ ersetzt eine Menge von Methoden wie z.B. „findById“ oder „findByName“ usw. Um die Methode „findBy“ zu implementieren kann man dafür z.B. die Reflection – Technik/Libraries einsetzten. Alle Repositorien Klassen erweitern die abstrakte Klasse „AbstractBaseRepository“, welche implementierte Methoden für die Datenmanipulation beinhalten. Ein Template „T“ wird bei der Vererbung mit den Modelklassen ersetzt. Aktuelle Realisierung des Repository Patterns ermöglicht spätere „Dependency Injection“ bei den Kontrollklassen in Package „Services“.

## 4 Package „Services“



Das Package „Services“ beinhaltet die Kontrollklassen, die später bei der z.B. GUI – Implementierung, sowie die anderen Entwürfe, eingesetzt werden. Ausgehend aus den Beschreibungen für die Anwendungsfälle, soll es für alle Aktoren des Auftragsdispositionssystems in erster Linie möglich sein, die zu bearbeitenden Daten, pflegen zu können, deswegen werden die Grundoperationen wie löschen, aktualisieren, erstellen, suchen (nach Id oder Nummer) in ein Interface „IBaseService“ zusammengefasst. Das Interface „IBaseService“ ist generisch entworfen, was bei der Vererbung von anderen Serviceinterfaces eine Einhaltung des „Don’t repeat yourself“ - Prinzip ermöglicht. Das Template „T“ wird wie bei „Repositories“ beim Erweitern von Serviceinterfaces mit Modelklassen überschrieben. Die „Child“ – Interfaces erweitern das „IBaseService“ mit den eigenen Methoden, welche in Serviceklassen implementiert werden.

### 4.1 Beispiel für den Einsatz bezogen auf Use Cases.

Hier wird anhand der ersten vier Anwendungsfälle aus der Ausarbeitung für die Use-Case-Analyse, die Verwendung der Entity- und Kontrollklassen (also Models und Services) kurz beschrieben.

#### 4.1.1 Auftrag Erfassen.

Der Verkäufer gibt in der geeigneten Eingabemaske die Suchkriterien für die Kundensuche. Dabei wird eine der Suchmethoden des „CustomerService“ z.B. „findByIdOrNumber“ aufgerufen. Wenn ein Kunde gefunden wird, wird dieses zurückgegeben, andernfalls wird eine null zurückgegeben was einen Grund zum grafischen „Alert“, dass Kunde nicht gefunden wurde, dient. Wird ein Kunde gefunden wird dieses in einem Customer – Objekt gespeichert, gleichzeitig werden die Daten des Kunden an die „View“ (Eingabemaske) weitergegeben und parallel zu der Property von Typ Customer, dem Orderobjektes zugewiesen. Der Verkäufer trägt weitere Auftragsdaten ein, die Daten werden dementsprechend den geeigneten Properties des Order-Objektes zugewiesen. Betätigt Verkäufer „Auftrag speichern“, wird die Methode „isOrderExists“ des Klasses „OrderService“ für die „if“ – Anweisung aufgerufen. Falls es sich um einen neuen Eintrag handelt, wird zu Property „state“ des Orderobjektes „OrderState.CAPTURED“ zugewiesen, Orderobjekt wird als Parameter an die Methode „addNew“ des „OrderService“ übergeben und somit wird neuer Auftrag erfasst und in Datenbank gespeichert.

#### 4.1.2 Kunde anlegen.

Der Verkäufer gibt in die geeignete Eingabemaske die Kundendaten ein. Die Daten werden zu geeigneten Properties des Customer-Objekts zugewiesen. Nachdem betätigen von „Kunde Anlegen“, wird die Methode „addNew“ der Klasse „CustomerService“ aufgerufen und das Customer-Objekt als Parameter, wird an die Methode übergeben. Ein Kunde wird in Datenbank/System gespeichert.

#### 4.1.3 Auftrag Suchen.

Nachdem der Verkäufer eine der Suchkriterien in die Suchmaske eingegeben hat, ruft das System eine der Suchmethoden der Klasse „OrderService“ auf. In dem Falle, dass Auftrag/Aufträge gefunden ist/sind, wird je nach Rückgabetyp ein Objekt von Typ „Order“ oder eine Collection von Orderobjekten zurückgegeben und an eine „View“ übergeben. Der Verkäufer bekommt somit eine Möglichkeit den gesuchten Auftrag zu öffnen.

#### 4.1.4 Auftrag freigeben.

Öffnet sich die Auftragsmaske, bekommt der Verkäufer eine Möglichkeit Auftrag zu korrigieren und

**Auftrag freizugeben. Klickt der Verkäufer auf „Auftrag freigeben“, wird dem Property „state“ ein Status „OrderState.APPROVED“ zugewiesen. Das Orderobjekt wird als Parameter an die Methode „update“ der Klasse „OrderService“ übergeben. Die Methode aktualisiert den Auftrag in System/Datenbank.**